RBUIS: simplifying enterprise application user interfaces through engineering role-based adaptive behavior

Conference or Workshop Item

RBUIS: Simplifying Enterprise Application User Interfaces through Engineering Role-Based Adaptive Behavior

Pierre A. Akiki, Arosha K. Bandara, and Yijun Yu
Computing Department, The Open University
Walton Hall, Milton Keynes, United Kingdom
{pierre.akiki, a.k.bandara, y.yu}@open.ac.uk

ABSTRACT
Enterprise applications such as customer relationship management (CRM) and enterprise resource planning (ERP) are very large scale, encompassing millions of lines-of-code and thousands of user interfaces (UI). These applications have to be sold as feature-bloated off-the-shelf products to be used by people with diverse needs in required feature-set and layout preferences based on aspects such as skills, culture, etc. Although several approaches have been proposed for adapting UIs to various contexts-of-use, little work has focused on simplifying enterprise application UIs through engineering adaptive behavior. We define UI *simplification* as a mechanism for increasing usability through *adaptive* behavior by providing users with a minimal feature-set and an optimal layout based on the context-of-use. In this paper we present Role-Based UI Simplification (RBUIS), a tool supported approach based on our CEDAR architecture for simplifying enterprise application UIs through engineering role-based adaptive behavior. RBUIS is integrated in our general-purpose platform for developing adaptive model-driven enterprise UIs. Our approach is validated from the technical and end-user perspectives by applying it to developing a prototype enterprise application and user-testing the outcome.

Author Keywords
Simplification; Adaptive user interfaces; Role-based; Enterprise applications; Model-driven engineering

ACM Classification Keywords
[Software Engineering]: D.2.11 Software Architectures - Domain-specific architectures; D.2.2 Design Tools and Techniques - User interfaces; [Information Interfaces and Presentation]: H.5.2 User Interfaces – User-centered design

General Terms
Design; Human Factors

INTRODUCTION
The functionality of software applications tends to increase with every release increasing the visual complexity. This phenomenon, referred to as "bloatware" [22], has a negative impact on usability especially for users who do not require the complete feature-set. Also, users have different layout preferences. Both feature-set and layout related choices could be affected by several aspects such as skills [30], culture [27], etc. This paper presents Role-Based UI *Simplification* (RBUIS), a mechanism for increasing usability by providing users with a minimal feature-set and an optimal layout based on the context-of-use (user, platform, and environment). We define a feature as a functionality of the software system and a minimal feature-set as the set with the least features required by a user to perform a job. An optimal layout is the one that maximizes satisfaction of the constraints imposed by a set of aspects. An optimal layout is achieved by adapting the properties of concrete widgets (e.g., type, grouping, size, location, etc.).

Feature-bloated *enterprise applications* are sold as off-the-shelf products to be used by people whose diverse needs in required feature-set and layout preferences are affected by multiple aspects. These applications serve various purposes in an enterprise's functional business areas (e.g., inventory, accounting, etc.). The literature clearly indicates that these systems suffer from usability problems. One example is given by a study carried out in the Nordic countries [20], which showed that almost 40% of the users find enterprise applications difficult to use to a certain extent. UI *simplification* could enhance the usability of these applications by catering to the variable user needs.

One method to achieve UI *simplification* is for enterprise applications to become adaptive/adaptable, respectively referring to the ability of tailoring software applications automatically/manually. Adapting a UI's feature-set could enhance user satisfaction [21] and make complex applications easier to use on mobile devices and by cognitively impaired users [16]. Also, adaptive/adaptable behavior has been used for tailoring the UI layout based on various aspects such as: "Accessibility" [24], "Platform" [7], "Natural Context" [6], etc. However, to meet enterprise application needs we propose the following criteria, for implementing UI simplification, based on the scale and complexity of these applications and the existing literature:

- providing a **scalable**, **extensible**, and **tool supported** mechanism capable of integrating in the development and post-development phases of enterprise applications and accommodating multiple adaptation aspects;

Authors' Version
EICS'13, London, United Kingdom, *Forthcoming*

- programming **role-based** adaptive behavior through both **visual** and **code** constructs hence allowing developers as well as I.T. personnel to define and reuse it;
- **preserving designer input** [26] on concrete UIs during adaptation instead of a fully mechanized UI generation;
- **reducing user confusion** [21] by proposing the adapted UI as a simplified alternative to the initial design rather than adapting it while the user is working;

We intend to meet the proposed criteria by using interpreted runtime models that allow more advanced adaptations and could be integrated as part of a generic solution offered as a service. The approach is based on our CEDAR architecture [1]. This paper makes the following contributions:

- An approach called Role-Based User Interface Simplification (RBUIS) composed of the following:
  - A mechanism for minimizing the feature-set at runtime by applying roles on task models
  - A mechanism for optimizing the layout by executing adaptive behavior workflows (visual and code-based constructs) on concrete UI (CUI) models
- *Cedar Studio*, our Integrated Development Environment (IDE) for devising adaptive model-driven enterprise UIs, provides tool support for our approach
- An evaluation of our approach with a set of studies based on two criteria: (1) technical feasibility and scalability, and (2) end-user satisfaction and efficiency

The remainder of this paper is structured as follows. We discuss the related work and briefly explain of our CEDAR architecture. Then, we elaborate on how RBUIS could be applied for minimizing a UI's feature-set and optimizing its layout based on CEDAR. Next, we provide an overview of our IDE *Cedar Studio*, and an example on building adaptive behavior models for use in our approach. Finally, we highlight the results of a study for evaluating RBUIS.

BACKGROUND AND RELATED WORK

This section briefly discusses existing work in terms of the four criteria we established in the introduction. We categorize existing work into feature-set minimization and layout optimization. These categories make up the *simplification* process and address the variable user needs in enterprise UIs. Additionally, we provide a brief overview of the CEDAR architecture based on which RBUIS is based.

Feature-Set Minimization

The *simplification* process should start by providing each user with a ***minimal feature-set*** to reduce unnecessary "bloat" [22] present in feature-rich enterprise applications.

Providing a multi-layered user interface design is promoted for achieving universal usability [28]. Other researchers propose using two UI versions, one fully-featured and another personalized, in bloated applications [21]. An earlier research work proposes the use of a "training wheels" UI that blocks advanced functionality from novice users [9]. These works present a sound theoretical basis, useful for providing the users of feature-bloated software applications with a minimal feature-set. Yet, the given examples, a basic text editor [28] and the Word 2000 menu [21], are not as complex as enterprise applications. Also, a generic, scalable, extensible, and tool supported mechanism is needed for applying feature-set minimization in practice.

Approaches from product-line (SPL) engineering [26] are used to tailor software applications and some particularly address tailoring UIs. MANTRA [7] adapts UIs to multiple platforms by generating code particular to each platform from an abstract UI model. Although SPLs can be dynamic [3], the SPL-based approaches for UI adaptation focus on design-time (product-based) adaptation whereas runtime (role-based) adaptive behavior is not addressed.

Role-based tailoring of the feature-set is sought after in commercial enterprise applications. Dynamics CRM 2011 [31] and SAP GuiXT [32] offer such a mechanism, yet it is not generic enough to be used with other applications and it requires maintaining multiple UI copies manually. Our approach is generic because it works at the model level.

Layout Optimization

Providing an ***optimal layout*** based on the context-of-use complements the *simplification* process. For example, SAP's usability (world's leading ERP [17]) is mostly affected by "Navigation" and "Presentation" [29] and its UI does not adapt to the user's skills [30]. Many existing works target the adaptation of the user interface layout, yet each uses a different approach to handle the adaptive functionality.

Fully mechanized UI construction has been criticized in favor of applying the intelligence of human designers for achieving higher usability [26]. It would be better if the designer could manipulate a concrete object rather than its abstraction [12]. Supple is a system for automatically generating UIs adapted to each user's motor abilities [16]. This automation prevents designer input on the concrete UI (CUI), which is the representation as concrete widgets (e.g., button, text box, etc.), making the system difficult to adopt for enterprise applications. Also, this approach has been criticized [24] for exceeding acceptable performance times.

Providing the adapted UI as an alternative version while maintaining access to the original full-scale UI, was shown to have higher user acceptance [21]. Yet, many platforms perform the adaptations while the UI is in use. MASP targets ubiquitous UIs in smart environments by adapting the UI whenever a context change is detected [6]. The MyUI platform also opts for adapting UIs while the user is working in order to prompt for user confirmation [24]. The choice of this adaptation mode is due to the ubiquitous nature of the target systems (e.g., Smart Homes). Since enterprise applications have a less ubiquitous nature with more complex WIMP style UIs, proposing the adapted UI as an alternative helps in avoiding confusion.

Scalability is important when targeting adaptive enterprise UIs. DynaMo-AID supports the development of adaptable context-aware UIs by generating what is referred to as a task tree forest [10]. As another work indicates [5], since each tree corresponds to a context's tasks, the combinatorial explosion makes the approach hard to scale.

The CEDAR Architecture

We created the CEDAR architecture [1] (Figure 1) as an approach for devising adaptive enterprise application UIs based on interpreted runtime models instead of code generation. The dynamic nature of these models gives more flexibility in performing UI adaptations and allows us to implement CEDAR as a **generic service oriented solution** that can be consumed by APIs using different technologies. These characteristics make CEDAR appropriate as a basis for our Role-Based UI Simplification mechanism (RBUIS).

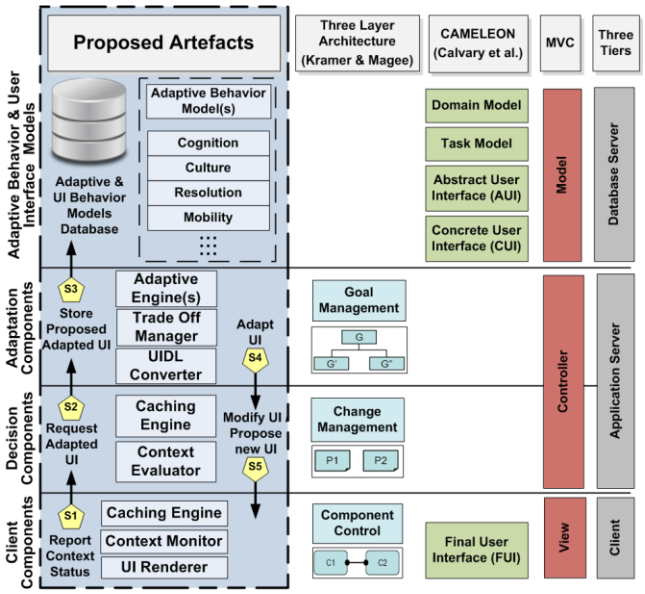


Figure 1. The CEDAR Architecture

We based CEDAR on the: (1) CAMELEON [8] reference framework (*UI Abstraction*), (2) Three Layer Architecture [18] (*Adaptive System Layering*), and (3) Model-View-Controller (MVC) paradigm (*Implementation*).

The coming sections show how RBUIS addresses the four criteria established in the introduction.

ROLE-BASED UI SIMPLIFICATION (RBUIS)

To simplify UIs, we need to provide a minimal feature-set and an optimal layout based on the context-of-use. The feasibility of adapting a single UI designed for the least constrained profile was demonstrated in previous research [15]. Our simplification mechanism will follow the same approach. In the case of the feature-set, the initial UI contains all the features hence it is without constraints. Yet, initial designer layout related choices (e.g., widget type, grouping, etc.) have to be the least constrained (e.g., in terms of screen size). The designer will devise the UI for the least constrained profile at design-time. Afterwards, a role-based approach is used to simplify the UI at runtime based on the context-of-use. Role-based modeling has been used for adapting the components of software applications [25], yet our approach is oriented towards merging access control with model-driven UIs to achieve UI simplification.

The standard for role-based access control (RBAC) could be utilized by enterprises for protecting their digital resources [13]. In RBAC, "Users" are assigned "Roles", which in turn are assigned permissions on "Resources". In our case the *users* are the enterprise employees logging into the system with their accounts, and the *resources* that we want to apply roles to, are the UI and adaptive behavior models. We merged the role-based approach with UI simplification to create Role-Based User Interface Simplification (RBUIS), in the spirit of RBAC. In RBUIS, *roles* are divided into groups representing the aspects based on which the UI will be simplified (e.g., literacy level, job title, etc.). RBUIS is applied after deploying the software in the enterprise. Managing this process could be a joint work between personnel from the software company in charge of the deployment process and the enterprise's I.T. personnel.

RBUIS comprises the following elements that support feature-set minimization and layout optimization:

Role-Based UI Models support *feature-set minimization* by assigning roles to task models (e.g., ConcurTaskTrees (CTT) [23]) to provide a minimal feature-set based on the context-of-use. This approach allows a practical realization of the concept of multi-layer interface design [28].

Role-Based Adaptive Behavior Models support *layout optimization* through workflows that represent adaptive UI behavior visually and through code. The adaptation is applied on the concrete user interface (CUI) models. Afterwards, adaptive behavior models are tied to roles to specify how the UI will be optimized for each set of users.

User Feedback for Refinement allows the users to reverse feature-set minimizations and layout optimizations, and to choose possible alternative layout optimizations. Keeping users involved increases their UI *control* [21] and *feature-awareness* [14] affected by adaptive/reduction mechanisms.

The following sections describe our approach in detail.

MINIMIZING THE FEATURE-SET

In order to minimize the feature-set we will rely on the concept of multi-layer interface design. This concept allows the users to control different sub-sets of the UI at any moment. For example, novice users could be given access to layer 1 and as they develop expertise could gain access to the upper layers at any time. RBUIS provides a practical approach for controlling the different UI layers. The meta-model for applying RBUIS on task models (CTT) is shown in Figure 2. CTTs were chosen to represent the task models due to their support of temporal constraints, which help in determining if simplifying a task could affect other tasks. Our approach in using temporal operators to check for task dependencies is similar to that of other researchers [4].

Feature-Set Minimization with RBUIS

Applying RBUIS on task models allows the minimization of the feature-set by revoking access to tasks based on roles hence achieving a role-based multi-layer interface design. Since we are initially designing the UI for the least constrained profile, the default policy will grant all roles access to all the tasks. This could be considered as a layer containing all the features. Afterwards, access could be revoked by allocating roles to tasks thereby creating separate layers, which users could gain role-based access to. Since users could be allocated multiple roles from the existing role categories, priorities will be used to provide enough flexibility to specify how roles override each other. Upon assigning the access rights to block tasks based on roles, a property (*concrete operation*) will specify whether to make a task invisible, disable it (keep data visible / protect data), or fade it until first use. The task model is mapped to the Abstract UI (AUI), which is in turn mapped to the CUI to hide, disable, or fade the relevant UI widgets.

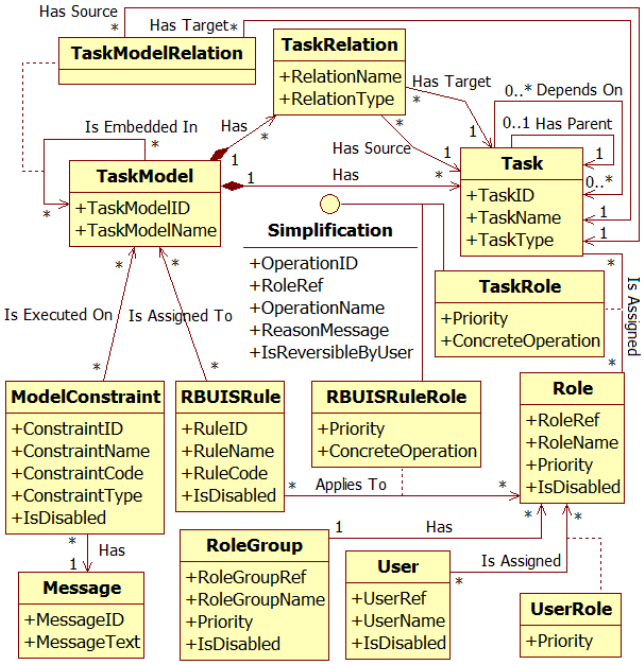


Figure 2. Meta-Model of Applying RBUIS on Task Model

Less Time Consuming Access Rights Allocation

Since enterprise applications encompass a large number of tasks that are used by hundreds of users, we need to make the allocation of access rights on the task models as little time consuming as possible. Traditionally, enterprise application users are allocated roles. This could be considered as a positive starting point. We will resort to the following features to minimize the time taken to allocate roles to tasks in the task models:

- A *default policy* grants access to all roles on all the application's task models hence making it only necessary to override this policy where access should be revoked. Each task will be implicitly allocated a fixed role called "*All-Roles*", which represents all the roles in the system and is granted access to execute the task. Access to the task will be revoked to all other explicitly assigned roles.
- Sub-tasks will *inherit* the access rights of the parent tasks while maintaining the ability to override these rights.
- In some cases the same functionality is replicated in many places within the application. Usually developers create visual components (CUI level) that could be reused in different places. By making task models *reusable* within one another, access rights allocated to a task model could roam with it whenever it is used again while maintaining the ability to override the initial rights. This feature is illustrated in Figure 2 with the recursive relationship "*Is Embedded In*" on the "*TaskModel*" class. Each embedded task model is connected to a source and a target task as shown on the "*TaskModelRelation*" class.
- *Rules* could be defined and applied to sets of task models based on each task's properties (ID, name, type, etc.). RBUIS rules are defined through our support tool (*Cedar Studio*) in the form of conditions using SQL syntax. Also, check lists are given to associate task models and roles with each rule. One basic example would be to revoke access to the role "*Cashier*" on all "*Interaction*" tasks with the words "*Enter Discount*" in the task name.

Applying RBUIS to Task Models at Runtime

Based on the CEDAR architecture, the UI models will be loaded on the server and the adaptive engine will apply RBUIS at runtime. To apply the concrete operations on the CUI, the Task Model is mapped to the AUI, which is in turn mapped to the CUI. A certain order should be followed to perform the elimination since each user could be allocated multiple roles simultaneously. The meta-model allows the assignment of priorities on different levels. The designer could specify where the priority is read from ("*RoleGroup*", "*Role*", "*TaskRole*", "*UserRole*"). Task-based assignments have a higher priority than rule-based ones unless specified otherwise. The following example demonstrates the process assuming the priorities were set at the "*TaskRole*" level:

- *UserA*: Novice, Manager
- *TaskX*: **1.** All-Roles (Allow) **2.** Accountant (Deny-Hide) **3.** Novice (Deny-Disable)

An excerpt of our algorithm is shown in Algorithm 1, the full version is included in a separate report [2]. Following this algorithm "*UserA*" is allowed to perform "*TaskX*" since "*Manager*" has the highest priority. In contrast, if "*Novice*" had a higher priority than "*All-Roles*", then "*UserA*" would have been denied access to "*TaskX*" hence disabling its CUI as indicated by the concrete operation.

The running time of our algorithm is estimated to be polynomial: $O\ (m \times (n \times l \times p \times (2\ j \log j + k) + n))$, where **m** = Num. of Task Models, **n** = Num. of Tasks in a Task Model, **j** = Num. of User Roles, **k** = Num. of Blocked CUI Elements for a Task, **p** = Num. of Parent Tasks for a Task, and **l** = Num. of Task Roles.

Algorithm 1. Feature-Set Minimization (Excerpt)

```
1. Simplify-Task (TaskID, UserRoles[ ], TaskRoles[ ], UIModel)
2.   foreach ur in UserRoles // Determine the Primary Role
3.     tr ← TaskRoles.GetRole(ur.RoleRef)
4.     if tr = null then tr ← TaskRoles.GetRole(All-Roles)
5.     ur.Priority ← tr.Priority;
6.   UserRoles.OrderBy(Priority)
7.   PrimaryRole ← UserRoles.First()
8.   if PrimaryRole.RoleRef ≠ All-Roles // Apply Concrete Operation to CUI
9.     blkdAUI←GetBlckdAUI(TaskID, UIModel.TMToAUIMap)
10.    blkdCUI←GetBlckdCUI(blkdAUI, UIModel.UIToCUIMap, UIModel.CUI)
11.    foreach element in blkdCUI
12.      switch PrimaryRole.ConcreteOperation
13.        case Hide: element.Visible ← false; break;
14.        case Disable: element.ReadOnly ← true; break;
15.        case Protect: element.ReadOnly ← true;
16.                      element.MaskChar ← '*'; break;
17.        case Fade: element.Opacity ← '30%'; break;
```

Model Checking using SQL

Since the access rights are being allocated by humans, model checking is needed to ensure that critical constraints are not violated. This allows our tool to issue appropriate warnings and errors. Several techniques exist for defining and evaluating constraints on models. For example, the Object Constraint Language (OCL) could be used to define constraints on UML diagrams. Furthermore, there are numerous tools that could be used for model checking (e.g., Z3, Spec#, Formula, etc.). In our case we need to define constraints on task models represented by CTTs. Since our approach is based on the CEDAR architecture, all the models are being stored in a relational database. This allows the model checking to be performed using SQL, which is more familiar to many developers and I.T. personnel than constraint languages such as OCL. The following example shows a constraint and its SQL-based solution in Listing 1.

Constraint: A sub-task should not be blocked for all the assigned roles because it will not be accessible by any user

Listing 1. Task Model Constraint Example using SQL

```
With SelTasks as (Select TM.TaskModelID, TM.TaskModelName, TK.TaskID,
  TK.TaskName From TaskModel as TM Inner Join TaskModelTask as TK On
  TM.TaskModelID = TK.TaskModelID Where TaskModelID in (@ModelIDs)),
  UserAccessOnTasks as (Select TaskModelID,TaskID, COUNT(case
  UR.CanExecuteTask when 1 then 1 else null end) as CanExecuteCount
  From SelTasks Cross Apply LoadSortedUserRoles(TaskModelID,TaskID)
  as UR Where UR.UserRolePriority = 1 Group By TaskModelID,TaskID)
Select SelTasks.* From SelTasks ST Inner Join UserAccessOnTasks UAT
  On ST.TaskID = UAT.TaskID and ST.TaskModelID = UAT.TaskModelID
  Where CanExecCount= 0
```

Constraints are defined in *Cedar Studio* and associated with task models through a system variable ("*@ModelIDs*"). Predefined functions such as "*LoadSortedUserRoles*" could be used in model constraints and extended when necessary. In this case the function loads the users and their assigned roles sorted by the priority of execution according to a certain task. The SQL statement would return the tasks that are violating the constraint, to be displayed on the screen.

Feature-Set Minimization Example

Although enterprise applications contain many complex examples, a basic example has been purposefully chosen in order to accommodate screen shots in the paper. Complex real-life examples were considered in our evaluation.

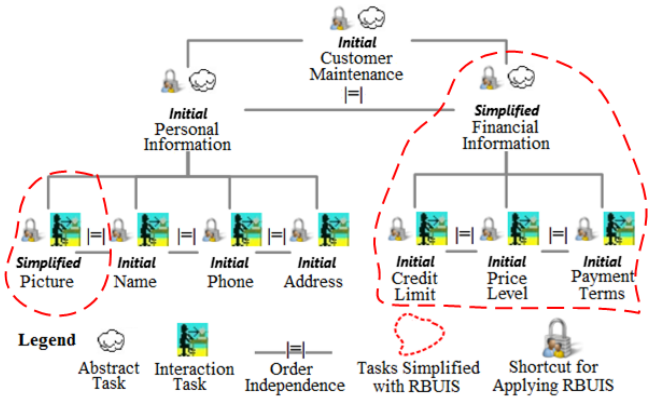
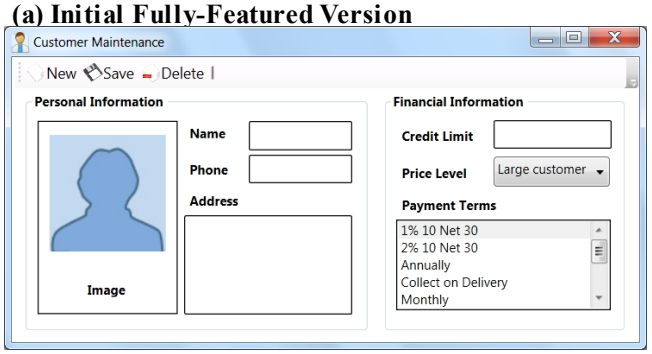


Figure 3. Simplified Customer Maintenance Task Model

The example illustrated in Figure 3 shows part of a task model built in *Cedar Studio* for a "*Customer Maintenance*" UI common in ERP systems. The lock-shaped buttons allow the application of RBUIS on any task. In this case, the tasks called "*Financial Information*" and "*Picture*" encircled in a dashed line are marked as **simplified** indicating that RBUIS has been applied. In the case of "*Financial Info.*" the access rights will get inherited by its sub-tasks. We considered a role called "*Cashier*" requiring a version of the UI showing only the "*Name*", "*Phone*", and "*Address*". This allows users working as cashiers to enter the initial information for a new customer on the counter without having to handle other details that could be added later. The initial version of the Final UI (FUI) is illustrated in Figure 4 (a), and the one simplified for the role "*Cashier*" is illustrated in Figure 4 (b). In this example, the concrete operation in RBUIS was set to "*Hide*" hence the widgets became invisible.

(a) Initial Fully-Featured Version



(b) Minimized Feature-Set Version for Role "Cashier"

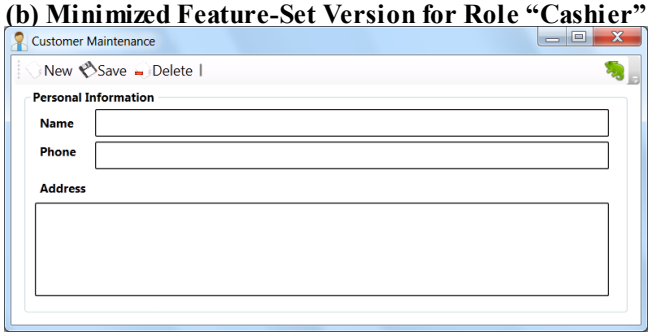


Figure 4. Feature-Set Minimization of Customer UI

OPTIMIZING THE LAYOUT

Providing users with an optimal layout could be based on various aspects (e.g., computer literacy, cognition, screen-size, etc.). In this section we present our generic mechanism for devising adaptive behavior for such criteria. Enterprise applications require an approach that allows developers as well as I.T. personnel to implement adaptive behavior. Our feature-set minimization mechanism allows RBUIS to be applied visually and through code-based rules. Similarly, our layout optimization mechanism allows the definition of adaptive behavior using a mix of visual and code constructs embedded in adaptation workflows. The meta-model for applying this mechanism on the CUI is shown in Figure 5.

Layout Optimization with RBUIS and Workflows

The representation of adaptive behavior has a great impact on the extensibility of any adaptive system. Most adaptive UI state of the art systems tend to employ an arbitrary design that hardcodes adaptation behavior within the software application, severely minimizing its reusability and extensibility. A graphical tool is suggested for hiding the complexity of defining UI adaptation rules [19]. This tool might not be able to handle all possible scenarios due to the limited use of a high level visual representation.

To balance between ease of use and flexibility, our approach combines high level adaptation operations and low level programming constructs by using both visual and code-based representations. Workflows are not strange to enterprise applications due to their use for devising customizable and reusable business rules that could be separated from the software code. With appropriate tool support, workflows could also provide visual programming constructs (e.g., control structures, error handling, etc.). Additionally, it is possible to define code-based adaptation operations that integrate within the visual workflow.

Our approach uses tool supported workflows, which could represent adaptive behavior with: (1) visual programming constructs, (2) compiled code libraries and dynamically interpreted scripts. The workflows are executed at runtime on the CUI models to perform the necessary adaptation.

To implement the workflows in practice we are using the Windows Workflow Foundation (WF), which is part of the .NET framework. WF provides a visual design tool, which we integrated into *Cedar Studio*. This design tool provides the ability to visually design activity workflows using a rich set of constructs, which could be saved in an XML-based format then reloaded and executed when an adaptation is needed. Furthermore, the supported constructs could be extended through external compiled class libraries developed in C# or VB.NET and dynamically integrated with our tool. We have used this capability to develop a construct capable of integrating within a workflow and executing non-compiled script code. We currently support Iron Python but other scripting or transformation languages (e.g., XSLT) could be integrated in the future.

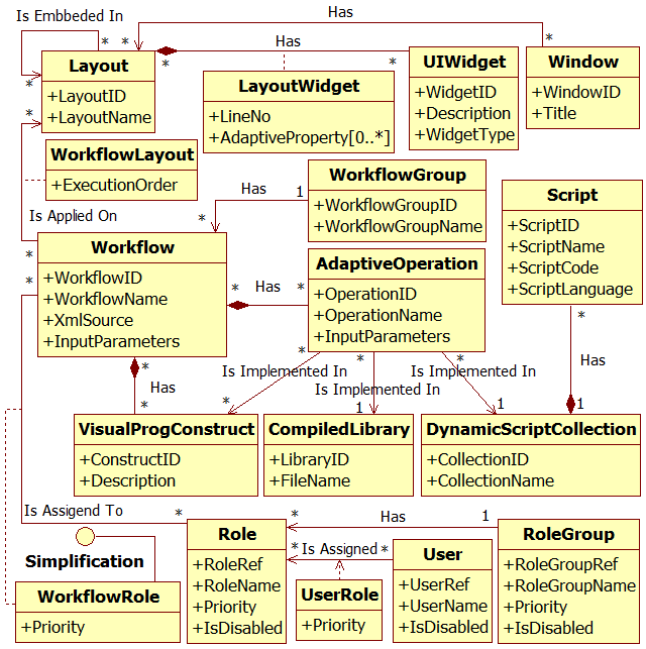


Figure 5. Meta-Model for RBUIS and Workflows on CUI

Applying RBUIS with Workflows at Runtime

Layout optimization is also based on our CEDAR architecture. After the feature-set is minimized, the workflows will be executed on the CUI by the adaptation engine. Afterwards, the FUI will be transferred to the client to be rendered on the screen. The process of selecting the workflows to be applied based on the user's role is illustrated in Algorithm 2 through an excerpt of our algorithm assuming the priority is read from the "*Roles*" class. The running time of our algorithm is established to be polynomial: $O\ (2\ m \log m + 2\ n \log n)$, where **m** = Num. of User Roles and **n** = Num. of Workflows to be Executed.

Algorithm 2. Layout Optimization (Excerpt)

```
1. Optimize-Layout (UserRoles[ ], Roles[ ], UIModel, LayoutID)
2.   foreach ur in UserRoles // Determine the Primary Role
3.     tr ← Roles.GetRole(ur.RoleRef)
4.     if tr = null then tr ← Roles.GetRole(All-Roles)
5.     ur.Priority ← tr.Priority;
6.   UserRoles.OrderBy(Priority)
7.   PrimaryRole ← UserRoles.First()
8.   WorkflowsToExecute[ ] ← GetWorkflows(PrimaryRole, LayoutID)
9.   WorkflowsToExecute.OrderBy(ExecutionOrder)
10.  foreach workflow in WorkflowsToExecute // Execute Workflows
11.    workflow.Execute(UIModel) // Execution Time Depends on Content
```

Layout Optimization Example

This example builds on the previous one illustrated in the feature-set minimization section. We consider two roles "*Sales Officer*" and "*Novice*". The "*Sales Officer*" requires the fully-featured UI illustrated in Figure 4 (a). The "*Novice*" requires layout optimizations that make functions accessible through on-screen buttons rather than a context-menu, and trading list boxes for radio buttons to fit more items on the screen. The workflow illustrated in Figure 6, represents the adaptive behavior by using three different techniques: (a) list boxes are substituted with radio button

groups using visual programming constructs, (b) function accessibility is set to high by calling an Iron Python script, and (c) the UI is refitted by calling a C# layout algorithm.

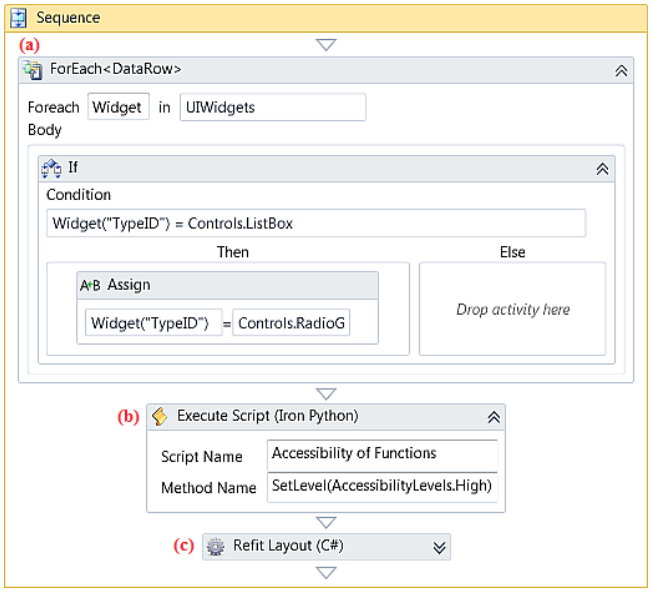


Figure 6. Layout Optimization Adaptive Workflow

The optimized UI in Figure 7 shows the functions for the image (add, remove, etc.) and address text-area (bold, italic, etc.) on the screen. In contrast, the version in Figure 4 (a) provided these functions through a context-menu. Also, the payment terms list box was substituted with a radio button group that displays more items on the screen. Some factors (e.g., access. of functions) are set by "*Adaptive Properties*" on the "*LayoutWidget*" class in the meta-model (Figure 5). In this case, the implementation of the adaptation behavior is part of the widget and it is just triggered from the workflow.

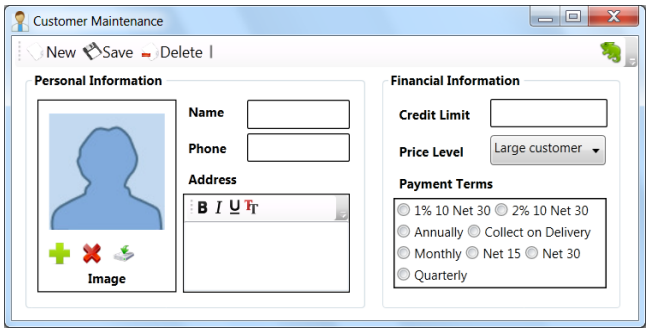


Figure 7. Optimized Layout of Customer FUI

USER FEEDBACK FOR REFINEMENT

Keeping the users involved in the adaptive process provides awareness of adaptive decisions and the ability to override role-based adaptations per user when necessary. In order to achieve this in practice we chose to transmit the final UI to the client with a list of the applied simplification operations. We denote such operations by the UML interface called "*Simplification*" shown in both meta-models. Our approach has two types of operations: *Feature-set minimization* and *layout optimization* identified by "*RoleRef*" and "*TaskID*" / "*WorkflowID*" respectively. The meta-model in Figure 2 shows "*ReasonMessage*" and "*IsReversibleByUser*" as attributes of the "*Simplification*" UML interface (same for Figure 5). These attributes indicate the reason behind the simplification and whether it its reversible by the users.

The users can click the chameleon icon in the top right corner of the UI (Figure 4 (b), and Figure 7) to show a list of the applied adaptation operations as illustrated in Figure 8. Afterwards, the users can uncheck any reversible operation (feature-set minimization or layout optimization) and apply the changes for one time only or for future use as well. Furthermore, layout optimizations have another feature that allows the users to choose from possible alternatives. This is achieved by assigning workflows to groups as shown in the meta-model (Figure 5). Workflows in the same group could serve as alternatives. For example, a group could encompass several workflows for adapting the selection widget (e.g., combo box, list box, radio buttons, etc.). After the user applies the changes, a request will be sent to the server to re-simplify the UI and exclude the operations that he or she unchecked. In case the user decides to keep the changes for future use, based on the CEDAR architecture, the changes would get stored and he or she will gain access to an alternative version of the UI. The example operations illustrated in Figure 8 are related to the simplified UI in Figure 4 (b). The operations inform the user that the UI parts pertaining to the *financial information* and *image* are unused by the user's role (*Cashier*) hence were eliminated. In this example, if the user unchecks both operations and applies the changes, the simplified UI in Figure 4 (b) will revert back to the original version in Figure 4 (a). If an operation is set as "irreversible by users" (e.g., due to security reasons) the check box would be disabled and a message would notify the user of the reason. If a feature depends on other disabled features, the user is informed that these features should be enabled as well. The dependency is determined from the CTT temporal operators and is defined on the meta-model (Figure 2) through the recursive relationship "*Depends On*" on the "*Task*" class.

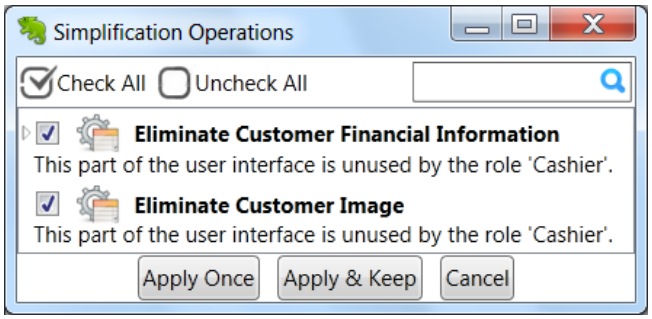


Figure 8. User Feedback - Simplification Operations

Even though in the case of feedback the UI is changing while the user is working, the user's initiation of the action reduces confusion due to the awareness and understating of the adaptation that is going to take place.

DEVELOPING APPLICATIONS WITH CEDAR STUDIO

Cedar Studio is our Integrated Development Environment (IDE) that supports the development of adaptive model-driven UIs for enterprise applications based on the CEDAR architecture. Due to space limits we will briefly describe its features in this paper. Interested readers could get more details from a separate report [2] and observe the tool in operation through online demo videos [33].

We created *Cedar Studio* in the form of an IDE to provide developers and I.T. personnel with an ease of access to all the visual-design and editing tools in one place. Currently, it supports visual design tools for: (1) Task Model, (2) Domain Model, (3) AUI Model, (4) CUI Model, and (5) Workflows. Also, it supports a combination of visual design and code editing tools for (1) Task Role Assignments and RBUIS Rules, (2) Model Constraints, and (3) Dynamic Scripts. One of the supported design tools (task model) is illustrated in Figure 9. Additionally, *Cedar Studio* supports automatic generation and synchronization between the various levels of abstraction (Task Model, AUI, and CUI) with the possibility to make manual changes at any of these levels.

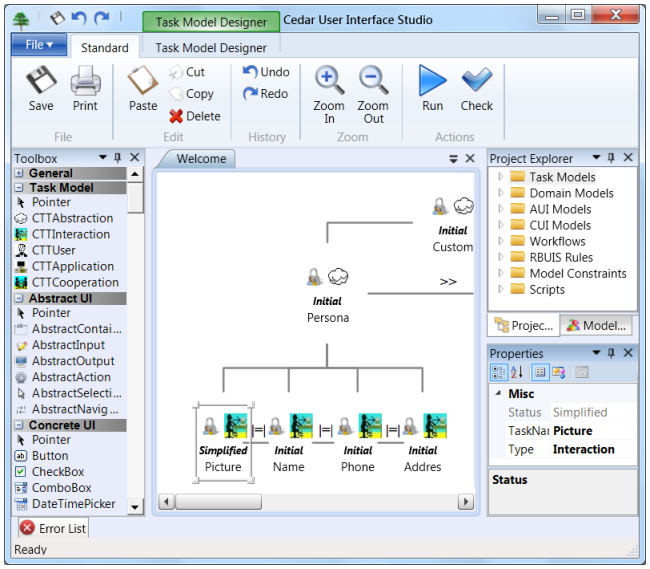


Figure 9. *Cedar Studio* - Our IDE Support Tool

Cedar Studio was designed as a tool for supporting our CEDAR architecture through UI and adaptive behavior models that would get stored in a relational database to provide easier runtime management and interpretation. The implementation of CEDAR is provided as a service that is consumed by *Cedar Studio* and technology specific APIs that allow enterprise applications to integrate with our solution. An API would include the client components illustrated in Figure 1. To test our approach we developed an API and a Toolkit in C# for the Windows Presentation Foundation. APIs for other presentation technologies (e.g., HTML, Java Swing, etc.) could be devised by anyone and used in combination with *Cedar Studio* for developing adaptive enterprise applications capable of benefitting from our simplification mechanism and any future extensions.

Adaptive UI behavior (e.g., widget hiding, substitution, etc.) could leave gaps and deformations in the layout, which are not esthetically desirable and could increase the navigation time (Fitts's Law). We required a mechanism to maintain plasticity, denoting the UI's ability to adapt to the context-of-use while preserving its usability [11]. Hence, we devised an algorithm to refit the layout based on its initial manual design by filling the gaps and adjusting the widgets' positions based on their new sizes and initial locations chosen by the designer. This technique creates a balance between fully automated approaches that generate the UI from an abstract model [16] and manual approaches that require developing and maintaining multiple CUI versions [32].

Cedar Studio is meant to be used during the development and post-development phases by developers, deployment teams, and I.T. personnel. The UI models are devised at the development phase and the simplification behavior could be added during the deployment phase according to the needs of each enterprise. This behavior could be based on user models such as the one described in the coming section.

BUILDING ADAPTIVE BEHAVIOR MODELS

One way to build adaptive behavior models for our system is to determine an aspect that influences enterprise application usability, statistically test its effect on UI alternatives, and implement the adaptive behavior for the alternatives using *Cedar Studio*. The outcome would be a general role-based adaptive model that could be refined by our feedback mechanism for particular tasks and users.

One such aspect discussed in the literature is "Computer Literacy" [29]. We setup a list of factors based on which the UI could be adapted and ran an online interactive survey to statistically test the effect of computer literacy on user preferences [2]. Although the list is not comprehensive it allows us to test our system against factors discussed in the literature and relevant to enterprise applications. We grouped the factors under categories that impact enterprise application usability ("*Presentation*" and "*Navigation*"):

Presentation: **Layout Grouping** (*Tab Page, Sub-Window, Group Box*), **Multi-Record Visualization** (*Grid, Carousel, Detailed Form*), **Simple Selection Widget** (*Combo, Slider, Radios*), **Multi-Record Input** (*Scrolling Grid, Non-Scroll. Grid, Form*), **Accessibility of Functions** (*High, Medium, Low*), **Information Density** (*High, Medium, Low*), **Text versus Graphics** (*Text Only, Image Only, Image & Text*)
Navigation: **Multi-Doc. UI** (*New Window, New Page, New Tab*), **Search the UI** (*Go to Widget, Filter, Filter & Re-layout*) **Navigation Structure** (*Menu, Tree, Panel*)

One should note that from a technical perspective adaptive behavior for all the factors is devisable using our platform. Yet, factors could vary based on different aspects. For example, our survey showed that computer literacy impacts "*Multi-Document UI*", "*Navigation Structure*", and "*Layout Grouping*", whereas "*Accessibility of Functions*" and "*Info. Density*" were shown to be impacted by culture [27].

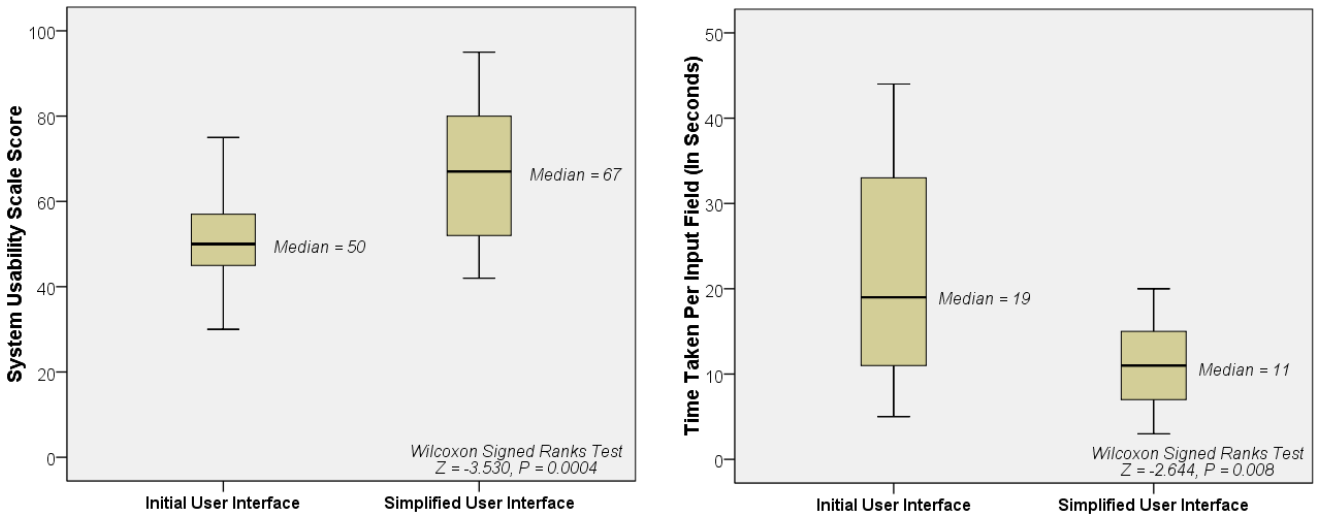


Figure 10. Evaluation Results for Role-Based UI Simplification

EVALUATING ROLE-BASED UI SIMPLIFICATION

Our simplification mechanism was evaluated [2] using an online interactive survey with a UI pair composed of an initial and a simplified UI. We selected the "*Customer Maintenance*" form of the SAP ERP. The initial version contains numerous nested tab pages and dozens of fields. Yet, users with different roles in the enterprise require a simpler version for managing basic customer records.

We developed a copy of SAP's UI alongside a simplified version containing the fields used to create a basic customer record. The fields were selected based on the variability in SAP's user needs [32]. The concrete operation was set to "*Hide*" with some fields being reversible by the user, and the widgets were regrouped accordingly.

Participants were asked to fill a set of fields required for creating a basic customer record using both UI versions. In the case of the simplified UI some of the fields had to be retrieved through the user feedback screen, allowing us to test how participants react to this feature.

In some cases, participants prefer the first UI option they see hence creating certain bias in a study's outcome. To avoid this potential bias we presented half of the participants with the initial UI first and the other half with the simplified one first. After each task, participants were asked to answer the System Usability Scale (SUS) questions, which allow us to detect usability differences between the two UI versions. Also, the time taken to complete each task was recorded.

We hypothesize that simplifying enterprise application UIs based on roles improves user satisfaction and efficiency.

The participants (n=25) never used the selected UI before. A Wilcoxon Signed Ranks Test showed that simplifying the user interface based on roles elicited a statistically significant improvement (Figure 10) in both SUS usability score ($Z = -3.530$, $P = 0.0004$) and task completion time ($Z = -2.644$, $P = 0.008$) hence confirming our hypothesis. The median SUS score was 50 for the initial UI and 67 for the simplified one. The median time taken to complete the task (seconds per input field) was 19 for the initial UI and 11 for the simplified one. The results were also reflected in the comments of some participants about the simplified version being more efficient whereas the initial UI made it complicated to locate fields. Also, the ease of use of the feedback mechanism was reflected by the fact that 80% of the participants were able to use it by only referring to a few words of instruction on its purpose.

CONCLUSIONS AND FUTURE WORK

We presented our Role-Based UI Simplification (RBUIS) approach, comprising feature-set minimization and layout optimization. RBUIS is based on our CEDAR architecture that is offered as a generic extensible service allowing the addition of adaptive behavior as needed. The scalability of our mechanism was shown by our complexity analysis.

Additionally, we introduced *Cedar Studio* our IDE that provides tool support for developing and maintaining adaptive enterprise UIs. We described how it can be used to represent role-based adaptive behavior visually (role assignment, and constructs in workflows) and through code (RBUIS rules, and compiled code/scripts in workflows).

Finally, we conducted a user study to evaluate RBUIS. The study showed a statistically significant improvement in user satisfaction and efficiency for simplified UIs. The outcome of the study also reflects the importance of a model-based approach that preserves designer input, made on the CUI, during adaptation. Also, by offering the UI as a role-based alternative our approach reduces confusion created by adaptations conducted while the user is working.

In the future we will extend our mechanism to support UI simplification in scenarios that require the use of multiple user interfaces for fulfilling a task. Additionally, more user studies will be conducted using eye-tracking in addition to measuring user satisfaction and efficiency.

ACKNOWLEDGMENTS

We would like to thank Prof. Helen Sharp and Dr. Sheep Dalton for their input on early drafts of this paper, Prof. Marian Petre for her comments on the video figure, and the anonymous reviewers for their valuable suggestions. This work is partially funded by ERC Advanced Grant 291652.

REFERENCES

1. Akiki, P.A., Bandara, A.K., and Yu, Y. Using Interpreted Runtime Models for Devising Adaptive User Interfaces of Enterprise Applications. ICEIS'12, SciTePress (2012), 72-77.
2. Akiki, P.A., Bandara, A.K., and Yu, Y. Cedar: Engineering Role-Based Adaptive User Interfaces for Enterprise Applications (2012). http://computing-reports.open.ac.uk/2012/TR2012-08.pdf
3. Bencomo, N., Sawyer, P., Blair, G.S., and Grace, P. Dynamically Adaptive Systems are Product Lines too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems. SPLC'08, Lero (2008), 23-32.
4. Bergh, J., Sahni, D., and Coninx, K. Task Models for Safe Software Evolution and Adaptation. TAMODIA'09, Springer (2010), 72-77.
5. Blouin, A., Morin, B., Beaudoux, O., Nain, G., Albers, P., and Jézéquel, J.-M. Combining Aspect-Oriented Modeling with Property-Based Reasoning to Improve User Interface Adaptation. EICS'11, ACM (2011),85-94.
6. Blumendorf, M., Lehmann, G., and Albayrak, S. Bridging Models and Systems at Runtime to Build Adaptive User Interfaces. EICS'10, ACM (2010), 9-18.
7. Botterweck, G. Multi Front-End Engineering. Model-Driven Development of Advanced User Interfaces, Springer (2011), 27-42.
8. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. A. Unifying Reference Framework for Multi-Target User Interfaces. Interacting with Computers 15, 3, Elsevier (2003), 289-308.
9. Carroll, J.M. and Carrithers, C. Training Wheels in a User Interface. CACM 27, 8, ACM (1984), 800-806.
10. Clerckx, T., Vandervelpen, C., Luyten, K., and Coninx, K. A. Task-Driven User Interface Architecture for Ambient Intelligent Environments. IUI'06, ACM (2006), 309-311.
11. Coutaz, J. User Interface Plasticity: Model Driven Engineering to the Limit! EICS'10, ACM (2010), 1-8.
12. Demeure, A., Meskens, J., Luyten, K., and Coninx, K. Design by Example of Graphical User Interfaces Adapting to Available Screen Size. Computer-Aided Design of User Interfaces VI, Springer (2009), 277-282.
13. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., and Chandramouli, R. Proposed NIST Standard for Role-Based Access Control. TISSEC, ACM (2001), 224-274.
14. Findlater, L. and McGrenere, J. Evaluating Reduced-Functionality Interfaces According to Feature Findability and Awareness. INTERACT'07, ACM (2007), 592-605.
15. Florins, M. and Vanderdonckt, J. Graceful Degradation of User Interfaces as a Design Method for Multiplatform Systems. IUI'04, ACM (2004), 140-147.
16. Gajos, K.Z., Weld, D.S., and Wobbrock, J.O. Automatically Generating Personalized User Interfaces with Supple. Artificial Intelligence, Elsevier (2010), 910-950.
17. Jacobson, S., Shepherd, J., D'Aquila, M., and Carter, K. The ERP Market Sizing Report. AMR Research (2007).
18. Kramer, J. and Magee, J. Self-Managed Systems: an Architectural Challenge. FOSE'07, IEEE (2007), 259-268.
19. López-Jaquero, V., Montero, F., and Real, F. Designing User Interface Adaptation Rules with T:XML. IUI'09, ACM (2009), 383-388.
20. Lykkegaard, B. and Elbak, A. IDC - Document at a Glance - LC52T. International Data Corporation (2011).
21. McGrenere, J., Baecker, R.M., and Booth, K.S. An Evaluation of a Multiple Interface Design Solution for Bloated Software. CHI'02, ACM (2002), 164-170.
22. McGrenere, J. "Bloat": The Objective and Subject Dimensions. CHI'00, ACM (2000), 337-338.
23. Paterno, F. Model-based Design and Evaluation of Interactive Applications. Springer-Verlag (1999).
24. Peissner, M., Häbe, D., Janssen, D., and Sellner, T. MyUI: Generating Accessible User Interfaces from Multimodal Design Patterns. EICS'12, ACM (2012), 81-90.
25. Piechnick, C., Richly, S., Götz, S., Wilke, C., and Aßmann, U. Using Role-Based Composition to Support Unanticipated, Dynamic Adaptation - Smart Application Grids. ADAPTIVE'12, IARIA (2012), 93-102.
26. Pleuss, A., Botterweck, G., and Dhungana, D. Integrating Automated Product Derivation and Individual User Interface Design. VaMoS'10, Universitat Duisburg-Essen (2010), 69-76.
27. Reinecke, K. and Bernstein, A. Improving Performance, Perceived Usability, and Aesthetics with Culturally Adaptive User Interfaces. TOCHI 18, ACM (2011), 1-29.
28. Shneiderman, B. Promoting Universal Usability with Multi-Layer Interface Design. CUU'03, ACM (2003), 1-8.
29. Singh, A. and Wesson, J. Evaluation Criteria for Assessing the Usability of ERP systems. SAICSIT '09, ACM (2009), 87-95.
30. Uflacker, M. and Busse, D. Complexity in Enterprise Applications vs. Simplicity in User Experience. HCI'07, Springer-Verlag (2007), 778-787.
31. Dynamics CRM 2011 - Role-Based UI. http://bit.ly/DynamicsRoleBasedUI.
32. GuiXT - Simplify and Optimize the SAP ERP UI. http://bit.ly/SAPGuiXTSimplifyUI.
33. Cedar Studio - Demo Videos. http://adaptiveui.pierreakiki.com.